National Research Council Canada
Conseil national de recherches Canada

Division of Electrical Engineering
Division de génie électrique

ERB-966

# *Using the Harmony Operating System*

W.M. Gentleman

December 1983
Revised October 1984, May 1985, March 1987

Canada

NRCC No. 27469

87-04579/

Copyright 1987 by
National Research Council of Canada

Permission is granted to quote short excerpts and to reproduce figures and tables from this report, provided that the source of such material is fully acknowledged.

Copyright 1987 par
Conseil national de recherches du Canada

Il est permis de citer de courts extraits et de reproduire des figures ou tableaux du présent rapport, à condition d'en identifier clairement la source.

Additional copies are available free of charge from:

Editorial Office Room 301
Division of Electrical Engineering
National Research Council of Canada
Ottawa, Ontario, Canada
K1A 0R8

Des exemplaires supplémentaires gratuits peuvent être obtenus à l'adresse suivante:

Bureau des publications, Pièce 301
Division de génie électrique
Conseil national de recherches du Canada
Ottawa (Ontario) Canada
K1A 0R8

# USING THE HARMONY OPERATING SYSTEM

W.M. Gentleman

ERB-966
NRCC No. 27469

DECEMBER 1983
REVISED OCTOBER 1984, MAY 1985, MARCH 1987

# TABLE OF CONTENTS


| | Page |
|---|---|
| ABSTRACT/RESUME | v |
| WHAT IS HARMONY? WHAT IS IT NOT? | 1 |
| WHAT IS REALTIME? | 2 |
| TASKS | 3 |
| COMMUNICATIONS AND SYNCHRONIZATION | 6 |
| STORAGE CLASSES | 9 |
| STREAM INPUT/OUTPUT | 10 |
| OTHER INPUT/OUTPUT INCLUDING INTERRUPTS | 12 |
| SERVERS AND CONNECTIONS | 16 |
| USER'S VIEW OF A PROGRAM | 20 |
| SUPPORT TOOLS | 22 |
| Listing | 22 |
| Examine | 22 |
| Bound | 22 |
| Debug | 23 |
| PERFORMANCE | 23 |
| ACKNOWLEDGEMENT | 24 |
| REFERENCES | 24 |
| APPENDIX A. A SAMPLE MULTITASK PROGRAM | 25 |
| APPENDIX B. A SAMPLE MULTITASK MULTIPROCESSOR PROGRAM | 29 |
| APPENDIX C. DECLARATIONS | 31 |
| APPENDIX D. SUPPLIED TASKS | 41 |
| APPENDIX E. SOFTWARE ENGINEERING CONSIDERATIONS: SOURCE MANAGEMENT AND THE ORGANIZATION OF THE HARMONY SOURCE TREE | 43 |

## ABSTRACT

Harmony is a multitasking, multiprocessing operating system for realtime control (such as is required in sensor-based robotics). It can also be used to implement "smart peripherals" to be used with itself or with some other operating system, and it provides a simple and inexpensive vehicle for experimenting with parallel computation. It is a portable system in that with moderate effort it can be realized on any computer of appropriate architecture. Our realization runs on the Chorus multiprocessor, a Motorola 68000 based Multibus machine, but realizations also exist for other 68000 family systems and for the Digital Equipment Corporation VAX. The operating system is mostly written in C, and user programs must currently also be written in C or Fortran. Programs written to run in Harmony, however, look very different from C programs written to run in Unix or some other conventional operating system.

## RESUME

Harmony est un système d'exploitation multitâche et multitraitement pour commande en temps réel (comme il est employé en robotique industrielle). Il peut aussi servir à la mise en application de "périphériques intelligents" qu'il commandera ou qui seront commandés par un autre système d'exploitation, et il permet d'effectuer de façon simple et à peu de frais des expériences de calcul en parallèle. Il peut être considéré comme un système portatif, puisque moyennant un effort modéré, il peut être appliqué a un ordinateur de type approprié. La version actuelle est utilisable sur le multiprocesseur Chorus, une machine Multibus basée sur le Motorola 68000 mais il y a des versions pour d'autres systèmes de la famille du 68000 et pour le Digital Equipment Corporation VAX. La majeure partie du système d'exploitation est écrite en langage C et, actuellement, les programmes d'utilisateurs doivent aussi être écrits en langage C. Les programmes écrits pour être traités par le Harmony ont cependant une apparence très différente de celle des programmes écrits en langage C devant être traités par le Unix ou un autre système d'exploitation ordinaire.

Copyright National Research Council of Canada, 1987

# USING THE HARMONY OPERATING SYSTEM

**W.M. Gentleman**

## WHAT IS HARMONY? WHAT IS IT NOT?

Harmony[1] is a *multitasking*, *multiprocessing* operating system for *realtime* control. (Each of these terms will be examined later.) It is a *portable* system, in that with moderate effort it can be realized on any computer of appropriate architecture. Our realization runs on the Chorus multiprocessor, a Motorola 68000 based Multibus multiprocessor using Omnibyte OB68K1A single board computers, but realizations also exist for other 68000 family systems and for the Digital Equipment Corporation VAX. The operating system is written in C (plus a small amount of assembler language) and user programs must currently also be written in C or Fortran, although the system is designed to support eventually other languages such as Pascal. Care has been taken to avoid dialect or library dependence on any specific C compiler — either the Unix[2] PCC portable C compiler or Whitesmiths' C compiler is used currently.

Although C is used as the programming language, programs written to run in Harmony look, and are thought of, very differently from conventional C programs which are written to run in Unix or some other conventional operating system. This difference goes beyond the question of what functions are available in the standard libraries and will be discussed at length later.

Harmony is an *open* system. This is a description that can be used at many levels. In the case of Harmony, this description is intended to imply that it is easy to use the system on many different configurations of hardware, and in particular to support peripherals not thought of when Harmony itself was originally designed. It is straightforward to integrate a new device into both the physical level of connection and the higher levels of software abstraction.

One major aspect is that Harmony is not designed to support a program development environment, and in particular it is not a multiuser timesharing system. Harmony devotes the entire resources of the machine to running a single multitask applications program, and is bound in with that program at link time (unless a system in ROM is used). Program development is done on some host computer, running its own operating system, and test executions are performed by downloading executable images into target processors. (Production programs instead might well be in ROM and be started by power up or reset.) In particular, Harmony programs cannot assume they have been started by a command interpreter like the Unix shell, and that they have command line arguments and redirected I/O. Indeed, they may not have terminal I/O at all.

---

[1] mark reserved for the exclusive use of Her Majesty the Queen in right of Canada by Canadian Patents and Development Ltd./Société canadienne des brevets et d'exploitation Ltée

[2] Unix is a trademark of AT&T Bell Laboratories

## WHAT IS REALTIME?

Returning to examine the terms used in the initial sentence, consider first the term *realtime*. Realtime programming has two independent aspects. First, programs must interact with the real world, and second, time itself may be a critical resource.

What makes interacting with the real world different from other kinds of programming is primarily that relevant events in the real world can happen independently of, and asynchronously with, the state of computation of the program. Conventional programming assumes a master/slave relationship between the computer and the rest of the world — things only happen when requested by the computer and are then guaranteed to happen, typically synchronously with the computation. This relationship does not hold for programs that interact with the real world. Events in the real world are often spontaneous, as seen by a program, and anticipated events often do not happen. Often the program cannot be aware of all of the relationships between events occurring in the real world. Indeed, the situations that can arise in the real world are often so varied that it is unrealistic to attempt to anticipate them all, and thus programs must cope with unexpected events.

Unless we are willing to limit programs to tracking a single sequence of events and to busywait for each event in the sequence to happen, we need a mechanism for expressing asynchronism in the program, so that the asynchronism in the program can model the asynchronism in the real world and be synchronized with it. This mechanism must be understandable, well defined, and efficient, so programs can be designed using it and can be reasoned about so that their behaviour will be known. The mechanism that Harmony provides is *tasks* (also called processes in some other systems). It will be described below.

A secondary aspect of interacting with the real world is that exotic peripherals are often required to sense or affect things of interest. Not only do these exotic peripherals require special command interfaces, because they do not fit the traditional I/O model, but they often require complex algorithms in order to be used, because they are flakey: sensors do not reliably return correct values, actuators do not reliably accomplish the desired effect. Error recovery is thus also intrinsic to interacting with the real world. Both error recovery and coping with unanticipated events are considerably simplified if resources can be allocated and deallocated dynamically. Such resources include tasks themselves, where instantiations must be created and destroyed dynamically, as well as blocks of memory of various sizes, which on demand are allocated to and freed by tasks, and connections between tasks and peripherals, which on demand are opened and closed by tasks.

Working with time itself as a critical resource is different from other programming in several ways. In conventional programming, time is irrelevant except perhaps in terms of overall efficiency considerations, or for recording and metering. Time can become a critical resource if, for instance, an external event requires a control signal in response within a guaranteed time, or if an external event produces volatile data which must be accepted soon enough to avoid the data being lost, or if periodic activity must be performed at regular intervals with little jitter, or if performance requirements indicate need for explicit parallelism between computation and external activities or between different computations, or if parallelism is intrinsic to the computation (as when stepping motors in orthogonal directions must operate simultaneously to produce a diagonal movement). Of course time is only a critical resource when the constraints are sufficiently stringent, given the available computational power, that conformance must be planned for and will not happen by accident.

Sometimes, when time is a critical resource, time is of the essence and actually defines or redefines the computation. For example, in the above case involving periodic activity, if new

sensor data are not available in time for the next cycle of computation, previous data must be extrapolated. Situations dealing with motion, and predicting trajectories, are often like this. More commonly, when time is a critical resource, the requirements translate into scheduling control. There is the choice of which computation to do first, when several are possible. When some computation becomes possible, there is the question of which current computations should be run to completion before starting this new one and which computations should instead be preempted. There can also be the necessity of ensuring the proper sequencing of otherwise independent computations. In Harmony, these things are achieved by dividing the program into tasks and using the rules for communication and synchronization between tasks, and the rules, including priority, which govern the scheduling of tasks on each processor. (These rules will be detailed below.)

One common characteristic when time is a critical resource is that no unanticipated delays in performing a computation are acceptable, that time performance must be guaranteed. This implies that no algorithms with execution time that cannot be bounded can be used, and apparently conflicts with the requirement from interacting with the real world that resources be dynamically allocated. The resolution of this dilemma is to preallocate those resources that will be required at critical times, doing the dynamic allocation at times which are not critical. For example, execution stacks are preallocated out of contiguous memory, rather than having stack frames allocated on demand from a heap, so that the cost of calling a function is bounded. Demand paging obviously is unacceptable in these circumstances.

Sensor-based robotics is an example of realtime computing that clearly illustrates both the real world aspect and the time critical aspect. In many robotic applications, there are a variety of objects in the robot's workspace. The robot must be aware of their positions, either to avoid them, or to make appropriate contact with them. Unless these objects are precisely located by mechanical fixturing, their exact locations, perhaps their shapes, and even their existence must be determined by sensors. Vision is one possible sensor, sensing the reactive forces on the robot's wrist is another, tactile and proximity sensing are yet others. The sensor measurements must be related to some abstract model of the environment for this data to be useful in planning robot motions. Note that this involves significant computation, using sophisticated algorithms. Some failure modes, and some reasons for unanticipated events, are physically obvious: lighting may be inadequate to visually resolve objects or objects may occlude each other, force exerted by the robot may cause objects to move or even break, proximity sensors may receive echoes from objects other than the one of primary interest. Having robots interact with objects in motion, even with another robot, is particularly challenging. The trajectory of the objects must be predicted, and must be continually updated as new sensor data become available. The time at which things are computed becomes important, it may be necessary to predict how long various computations will take, and computations may have to be scheduled so that values will get computed soon enough. Not only does it become necessary to know how long it takes for sensors to obtain data, and for the computer to analyze the data, and for the robot and other effectors to perform operations, but time becomes of the essence in a program, such as having to specify when a robot arm should be at some point, and what speed it should have in what direction, in order to intercept some object, match the object's motion, and connect with it.

## TASKS

At several points in the foregoing discussion, the concept of tasks has been alluded to, and it is now appropriate to define it. Tasks are a concept not used in conventional programming, not even in ordinary C programs in Unix, but they are the heart of programming in Harmony. A program in Harmony is made up of one or more, but typically many, tasks. A task is best thought of as like a subroutine, except that an instantiation of it must be explicitly created, and

once it has been created, it executes independently of, and in parallel with, the task that created it. (Actually, tasks that are completely independent of each other are not of very much use, so a mechanism must be provided for tasks to communicate and synchronize with each other. The mechanism that Harmony provides is message passing, which will be described shortly.) Each task itself executes sequentially and synchronously, like a conventional program — the asyn - chronism and parallelism are strictly between tasks.

Executing in parallel is, of course, only actually possible when the computer is a multi - processor, and the tasks are executed by different processors. There are two classical reasons for using multiprocessors: to increase total computing capacity, and to have redundancy to provide reliability in case of hardware failure. In realtime systems, there is a third reason: to guarantee responsiveness, by ensuring that a processor will be available to service an event soon enough. Harmony thus, as stated at the beginning, supports multiprocessing. When a task is created, the processor on which it will execute is specified, and this need not be the same as the processor on which the task requesting the creation is executing. A task competes for scheduling only with other tasks that run on the same processor. Note that the Harmony model of tasks tied to specific processors is different from the model used in some other systems, where any processor can execute any task. The Harmony model recognizes that processors may have local resources, such as memory or peripheral IC's, that can only be accessed by tasks executing on that processor, or that are much more efficient if so accessed. It also facilitates different processors executing different instruction sets. The hardware assumptions Harmony makes are discussed elsewhere (1).

Some simple multiprocessing systems only allow a single task to execute on each proces - sor. This has the advantage of simplicity and makes context switching overhead negligible. However, it has significant disadvantages. The number of tasks can no longer represent the natural asynchronism of the computation but is constrained by the number of processors in the configuration, and failed processors must be allowed for. Moreover, processors, though cheap, are not free, and utilization may be poor when it is known that certain tasks will execute at disjoint times. Accessibility of peripherals can force several tasks to be executed by the same processor. Certain operating system services, required by each processor, are most conven - iently implemented by separate tasks executing on that processor. Consequently, Harmony supports multiple tasks on each processor.

Since multiple tasks can be created to run on any given processor, Harmony must provide multitasking (sometimes called multiprogramming) to share the processor amongst these tasks. Timesharing systems typically do this with timeslicing and round-robin scheduling, but this would be inappropriate for realtime, given the requirements for control of scheduling as men - tioned above. Instead, each processor supports a separate priority scheduling system. Each task has a fixed priority level. There is a separate FIFO ready queue for each priority level, and all tasks of that priority level which are not blocked are in that queue (blocking actions will be described later). The first task on the highest priority nonempty queue is the active task, i.e., the one which is executing. It continues to execute until it blocks or is preempted by a higher priority task becoming ready. That is, Harmony tasks run *natural break* and support preemp - tion. Priority, as used in realtime systems, does not refer to *importance*, but rather to *urgency*. The ideal effect of priority is that the scheduling behaviour of a task can be understood by understanding only those tasks at the same or higher priority levels — the existence of lower priority tasks, much less their behaviour, can be ignored. In particular, if there is only one task at the highest priority, strict bounds can be placed on scheduling delays and jitter. Another important effect of priority is that nonblocking actions of high priority tasks can be viewed as atomic by lower priority tasks.

A task is thus the unit of sequential and synchronous execution. A task is also the unit of resource ownership. All resources, such as memory blocks, peripherals, or I/O connections, are owned and managed by some task or other.

For a task to be created, there must be a corresponding task template, which specifies the essential parameters of the task. (Multiple instantiations of the task defined by a given template can exist at the same time.) A task template is the following struct:

```
struct TASK_TEMPLATE
 {
  unsigned GLOBAL_INDEX;
  void     (*ROOT)();
  unsigned STACKSIZE;
  unsigned PRIORITY;
  unsigned LOCAL_TASK_MANAGER;
 }
```

The fields in this struct are as follows:

GLOBAL_INDEX — is an integer identifying this template. It must be unique over all templates on all processors for this program. The range of these indices is 1 through N_task_templates, a ROM_extern configuration parameter for which 200 is usually a satisfactory value. It is wise to use only larger values, say over 50, and also to use symbolic constants instead of literal integers. The indices 1, 2, and 3 are reserved: 1 is the first user task (i.e., it calls procedure main() ), 2 is the directory task (i.e., it provides task id's corresponding to symbolic names), and 3 is the gossip task (i.e., it provides a general reporting and logging mechanism for other tasks). The system itself creates on processor 0 one instance of each of these three tasks.

ROOT — is the function that is the code which the task will execute. (This function can, of course, call others.) It is a normal parameterless function with no return value. If the function ever returns the task is destroyed; however, the parent is not notified. More than one task template might have the same root function.

STACKSIZE — Each task has its own stack, which is preallocated when the task is created. Unfortunately, many compilers, including Whitesmiths', do not generate code to check for stack overflow. A program development tool, the stack bounder, helps with the choice of this value.

PRIORITY — Task priority is specified by an integer; larger is lower priority. When Harmony is ported to a given target, these priorities are arranged to be consistent with whatever hard - ware priorities the target implements. For instance, on the Chorus multiprocessor, values 0 through 4 correspond to the Motorola 68000 hardware interrupt levels 5 through 1. The

number of priority levels is not fixed, but having too many levels gains little and adds overhead.

LOCAL_TASK_MANAGER — is a field stuffed by the system. It identifies the local task manager that creates and destroys instances of this task. Since each processor has its own local task manager, this effectively determines which processor executes these instances.

The executable image for each processor is compiled and linked separately in Harmony. For each processor, that image is defined, in part, by the declaration of a vector of task templates, _Template_list, that determines what tasks can be created on this processor. The vector is terminated by a null task template, i.e., one in which all fields are zero. Tasks are created by calls to the _Create primitive, specifying a template index.

```
id = _Create( global_index );
```

The _Create primitive returns an unsigned id, which is used to identify this particular instantiation in future, e.g., for message passing, task destruction, etc.

Two functions are provided to obtain other id's that are often wanted: the id of the task itself and the id of the task which created this one.

```
id = _My_id();
id = _Father_id();
```

As described above, a task is destroyed if its root function returns. It can also destroy itself by calling the _Suicide primitive.

```
_Suicide();
```

More generally, any task can be destroyed by calling the _Destroy primitive.

```
_Destroy( id );
```

Destruction implies that the task is stopped, its id made invalid, and that its resources are returned to the system. Its connections (which will be discussed later) will be closed. Destruction of a task also implies destruction of all the descendants of the task, i.e., those tasks it created, those they created, etc. This is important in order to avoid the existence of orphans, tasks that hold resources but are useless because no other task knows of their existence. The _Destroy primitive is synchronous, which means that all this will be completed before the primitive returns, so the resources are immediately available for reuse.

## COMMUNICATIONS AND SYNCHRONIZATION

Although the main purpose of using tasks is to allow them to proceed with their own computations independently and in parallel, each running at its own speed, in most practical programs the tasks need to communicate and synchronize with each other occasionally. In particular, if more than one task accesses the same location in storage, and if the value at that location can change, then there is a possibility of a critical race where the result of the program depends on when which task reads and when which task writes the value. Various systems

have provided different mechanisms for dealing with this problem (2 – 4), including sema - phores, monitors, conditional critical regions, rendezvous, and several kinds of message passing. Harmony addresses the problem by a paradigm for structuring multitask programs, using four primitives that provide message passing. A *message*, in Harmony, is a variable length contiguous block of storage, the first two bytes of which are a 16 bit unsigned integer specifying the length. Typically the type of the message is actually a struct or union, but the primitives declare the type for their arguments as pointer to char, requiring a cast. The primitives are implemented as functions:

```
id = _Send( rqst, rply, id );
id = _Receive( rqst, id );
id = _Try_receive( rqst, id );
id = _Reply( rply, id );
```

The semantics of these primitives are as follows: A task wanting to send a message to another task sets up that message in space pointed to by the rqst argument, sets up space pointed to by the rply argument into which to receive a message in reply, and then calls the _Send primitive specifying in the id argument the id of the desired correspondent task. A task wanting to receive a message must set up space pointed to by the rqst argument into which the contents of the sender's rqst message can be copied, and then calls either the _Receive or the _Try_receive primitive, specifying in the id argument either the id of a particular corre - spondent task or 0, the latter indicating that a message will be accepted from any task (in fact FIF0). When the _Receive or _Try_receive primitive returns, the value id returned by the function is the id of the correspondent task, and the sender's message will have been copied into the receiver's message. After the receiving task has finished processing the sender's request, it sets up, in space pointed to by the rply argument, the message to be replied back to the sender, and calls the _Reply primitive with the id argument specifying the sender. Speci - fying the sender is necessary because it is not required that a sender be replied to before receiv - ing another message, nor that messages be replied to in the order they were received, in the inverse order to which they were received, or any other fixed order. A reply can only be made to a task from which a message has been received and to which a reply has not yet been made, however. The message being replied to the sender is copied from the receiver's space to the space provided by the sender before the _Send primitive returns.

A particularly important aspect of the semantics of these primitives is their blocking behav - iour, for this is how tasks synchronize with each other. The sender blocks from when the _Send primitive is called, not just until the corresponding _Receive or _Try_receive primi - tive is called by the receiver but until the receiver calls the corresponding _Reply primitive. A receiver calling the _Receive primitive blocks if the id argument is 0 and no task has already called the _Send primitive to send a message to this receiver, or if the id specifies a particular task and that task has not yet called the _Send primitive to send a message to this receiver. The receiver calling the _Receive primitive unblocks when the message for which it is waiting becomes available. By contrast, the _Try_receive primitive is nonblocking and fails immedi - ately, returning 0, if the desired message is not available when the receiver calls the primitive. The _Reply primitive is a nonblocking operation for the receiver (because the sender is known to be already blocked) and it also unblocks the sender, allowing it to run again. Of course if the sender and receiver happen to run on the same processor, the normal scheduling rules will dictate which actually runs first. Nothing else in the foregoing discussion is affected by whether the communicating tasks happen to be on the same processor or different processors.

Because tasks can be destroyed, it is possible that the primitives can fail. This is the reason _Send and _Reply are also implemented as functions: all four primitives return the id of the correspondent task if they succeed, and 0 if they fail. Another point to note is that the length of the message indicated in the space copied from and copied to may disagree. Only the shorter length is actually copied, but since the length copied from is the first thing copied into the space copied to (overwriting the length originally there), it is possible to find out if the message was truncated, and how much space would be required to get it all.

There is a certain symmetry between _Send and _Reply in that both transfer data to another task. By turning around the usual role of these primitives, a nonblocking communication is possible between one task and another which has prearmed for the communication by sending to indicate it is ready to receive a message. That is, suppose task A has some data to transfer to task B. Normally, one would think of A sending a message to B. If this is done, A will block until B replies. If, however, it is important that A not block, the communication could be reversed and B could instead send to A and block. When A is ready with the data, it can transfer the data to B in a reply, with which A will not block but will continue to execute, and B will also resume execution in parallel. When B finishes processing the data, it can send to A again, returning any results and preparing to repeat the cycle.

The paradigm (2) is that, in general, tasks do not use resources directly; for example, several tasks will not all access a location in memory that can change. Instead, each such resource is *owned* by a single task, and operations on the resource are performed by that task on behalf of other tasks that have sent messages requesting these operations. The task given the responsibility for some resource is referred to as a *server*, since it is seen by other tasks as providing a service — the tasks using this service are referred to as *clients*. Mutual exclusion follows directly from the fact that the server is a single task, with a single thread of execution, and messages are received one at a time. Sometimes, parallelism can be exploited with respect to the resource the server controls. This is done by the server having worker tasks that employ the reverse communication described earlier — the worker tasks send to the server whenever they have completed the last item of work they have been given, and consequently are available to be given (by a reply) another item of work to do.

Occasionally, it is appropriate for closely cooperating tasks to share memory, because the protocol by which they cooperate ensures this is safe, and because the overheads of passing the data back and forth by message passing would be excessive. (Large constant tables are a classic example of where sharing is necessary and safe.) Harmony supports this, in that addresses mean the same to all tasks on the same processor, so pointers can be passed in messages and used by recipient tasks. When Harmony is implemented in a flat address space shared memory multiprocessor, this extends to pointers used by all tasks, whether on the same processor or not. However in a *thinwire* implementation, where processors communicate via reliable links but cannot directly access each other's memory, a pointer is only useful within a single processor. Because processor images are linked separately, externals are shared among tasks on the same processor, but only among tasks on that processor. In general, sharing memory requires great care and understanding, however, and should be avoided where possible. The tasks supplied with Harmony do not use shared memory, except, of course, in the debugger, and in a few cases where it has been necessary to provide a readable copy of a write-only device register that unfortunately must be shared by otherwise independent tasks.

The semantics of the call to _Send used by a client task to a server are identical to a procedure call, and so are familiar and comfortable to all programmers. Indeed, this semantics has become known in the literature as the remote procedure call. Because _Send and _Receive can block, there is some possibility of deadlock, but in practice it does not happen when the client/server paradigm is used because almost the only way to get a deadlock would

be to have a cycle of tasks sending to each other. With a hierarchy of services implemented using lower level services, that never happens.

## STORAGE CLASSES

A program in C running under Harmony has access to five storage classes: external, static, auto, register, and dynamically allocated.

The first two can be useful, but have complicated aspects. Because the executable image for each processor is linked separately, externals or statics referenced by tasks on one processor are different from those referenced by tasks on another, even though the name in the source code is the same. On the other hand, all tasks on the same processor will access the same instance of a named external or static — which, unless the external or static is initialized and not subsequently assigned to, can lead to critical races as noted previously. It is also worth noting that ROMability is often an objective of realtime systems, and that an external or static that is initialized by compile-time initialization is not ROMable if it is subsequently assigned to.

The situation is much simpler for variables of the auto storage class. They are allocated on the stack of the task, dynamically as the task executes. There is a different instance at each function call, never mind for each task. The stack for the task itself is dynamically allocated when the task is created and recovered when the task is destroyed. Because local memory typically has faster access, the stack for a task is allocated from memory local to the processor on which the task will run.

Some of the hardest bugs to find involve stack overflow, where the stack required at some point in the nesting list is bigger than what was allocated for the task. Unfortunately, there is rarely adequate hardware checking for this when there are multiple stacks in the same address space, and many compilers, including Whitesmiths', do not generate code for a runtime software check on function entry. Harmony provides two aids to alleviate this unsatisfactory situation. The first is a function that can be called to determine if the stack for this task is already overflowed,

```
boolean = _Stackoverflow();
```

The other is a program development tool, bound, that determines the required stack size when the call graph for a task is simple (an acyclic digraph of known functions) and interactively aids the user in choosing an appropriate stack size when there is recursion or indirect function calls.

The register storage class of course has no problems for tasks.

Storage can be dynamically allocated in that a contiguous block of storage is obtained by a call to the primitive

```
ptr = _Getvec( size );
```

where size is the number of bytes in the desired block, and the value returned is typed as pointer to char, so typically must be cast to the required type. The value 0 is returned if the allocation failed. Memory is allocated from a pool local to the processor executing the primitive. The memory obtained is not initialized to any particular value.

When a block of storage is no longer required, it can be returned to the storage pool by calling the primitive

```
_Freevec( ptr );
```

When a task is destroyed, all of its allocated blocks are automatically returned to the storage pool.

Occasionally it is desirable to allocate an overly generous block of storage, then return the excess to the storage pool once the true requirements are known. Reducing the block to size and returning the excess is achieved by calling the primitive

```
_Trimvec( ptr, size );
```

The size of a dynamically allocated block is available from the primitive

```
size = _Sizeof( ptr );
```

A first fit storage allocation strategy is used for _Getvec because, although it does not provide the fastest allocation, the probability of denials is reduced since the *graded* distribution of idle blocks that develops over time preserves larger blocks. The classical problem with first fit is that each search for a suitable idle block examines whatever permanently allocated blocks are at the beginning of the pool as well as the blocks that are too small. Storage allocation for blocks size or larger may be faster if from time to time the starting point for the space search is reset by calling the primitive

```
_Tune_Getvec( size );
```

## STREAM INPUT/OUTPUT

The model of stream I/O is a powerful and uniform abstraction that is useful for doing input/output to a wide range of devices, from serial lines to terminals to tapes to disk files. A stream is an infinite sequence of bytes in which a current position is defined, and on which the operations of getting the next byte, putting the next byte, and changing the current position may be performed. Any higher level structure must be inferred from the value of the current byte, the context of the current byte, or the current byte position. For example, numbers are formed as sequences of bytes separated by delimiters; variable length lines of text are readily supported by separating the lines by a new-line symbol; end of file is indicated by an end-of-file symbol; and transparent communications line control is achieved by ASCII control characters and escapes inserted into the text to be transmitted.

Harmony supports *connection-oriented* stream I/O. (All types and manifest constants referenced below are declared in the standard header, sys.h, which should be included in any compilation of a Harmony program.) A stream is made accessible by being opened by a task,

```
ucb = _Open( directive, userid );
```

where directive is a string that symbolically defines the stream of interest and will be dis - cussed more fully later, and userid is an unsigned which is an application defined identifica - tion that servers may use to enforce access control. The value returned, ucb, points to a dynamically allocated struct called a *user connection block* that defines the server and connec - tion number associated with this stream and points to the ucb_xtra, a second dynamically allocated struct that contains the buffer and supporting pointers. The user of stream I/O need

not be aware of these details. The value 0 is returned if the open failed. There is no fixed limit as to how many streams may be open. In general, the same stream may be opened more than once (although some servers may disallow this), each with its own current position. This is typically done when several tasks use the same stream.

A stream which no longer needs to be accessible should be closed:

```
_Close( ucb );
```

Destruction of a task automatically closes all its open streams. It should be noted that closing a stream does not imply flushing any buffers (see below) and this must be done explicitly.

For each task, one of the opened streams is selected for input at any time, and all input comes from this stream until the selection is changed by

```
prev_ucb = _Selectinput( ucb );
```

Similarly, for each task one of the opened streams is selected for output, and all output goes to that stream until the selection is changed by

```
prev_ucb = _Selectoutput( ucb );
```

In this way, input bytes can be obtained from the selected input stream without the cost and inconvenience of specifying the stream each time

```
byte = _Get();
```

and similarly output bytes can be put to the selected output stream without the cost and inconvenience of specifying the stream each time

```
byte = _Put( byte );
```

The byte output is returned as the value of _Put because of the convenience this often provides in program control structure.

Although the abstraction of stream I/O is attractive, it would be impractically expensive if it were not actually buffered, so that _Get and _Put usually just reference a local buffer instead of actually doing I/O or even sending requests to a server. This buffering is transparent on input (although finding whether a byte is available for input is impossible without risking blocking) but not on output. The output buffer will be sent to the server to be output when it is full, but situations often arise where the buffer should be forced out without waiting to be full. This is done by

```
_Flush();
```

(It is unfortunate that the phrase *flush the buffer* has two conflicting interpretations — instead of the one used here, there is a different one sometimes used in other systems to imply that the buffer contents should be discarded.)

The buffered input makes possible a function which is very convenient when breaking input tokens at delimiters, or when reading input that may be of several different forms. This is the action of putting the last byte read back into the input stream, so it will be read again by

the next _Get. This action only works for the most recently read byte on the current input stream, although that can be any input stream.

```
_Unget();
```

For a more general mechanism, it is meaningful for some streams, such as sequential disk files, to change the current position to an arbitrary position in the stream, either indexed absolutely from the origin of the stream, or relative to the current position.

```
_Seek( ucb, position, relative );
```

Here position is a long int, and relative is nonzero if the indexing is relative to the current position.

On top of this basis, a few higher level functions are provided for convenience. Numbers are read free format, with the C convention where 0xNNN represents hex, 0NNN octal, and NNN with nonzero leading digit decimal, by

```
number = _Getnum();
```

Numbers can be printed in decimal by

```
_Putdec( number );
```

or in hexadecimal by

```
_Puthex( number );
```

and strings can be printed by

```
_Putstr( string );
```

This can all be combined by

```
_Printf( format, item1, item2, ... );
```

which prints the variable number of items according to the format string. As in Unix, the format characters are directly output, except that %% is printed as %, %c prints the corre - sponding item as a character, %i prints the corresponding item as a decimal integer, %h prints the corresponding item as a hexadecimal integer, and %s treats the corresponding item as a pointer to a string and prints that. Full format control as in Unix is not supported. In particular, there is currently no floating point conversion.

## OTHER INPUT/OUTPUT INCLUDING INTERRUPTS

Not all I/O devices fit the stream I/O abstraction. Moreover, for some I/O devices, even though the abstraction can be forced to fit by embedding control in the stream of bytes, there are realtime applications that cannot afford to pay the performance penalty of such a uniform implementation. Consequently, it is necessary for Harmony to allow the user to design and implement his own I/O abstractions, and for tasks in a user program to have the ability directly to perform I/O operations and to handle interrupts. Even when the stream I/O abstraction is

used, these facilities may be needed because, Harmony being an open system, the user may have unique devices, for which he must therefore supply the tasks supporting the streams. In any case, the user will have to specify for any particular program which of the available servers are needed, because configuration of a control computer for realtime applications varies so widely.

To allow the user program to do I/O directly, with maximum performance, Harmony ignores the supervisor versus user state distinction present in most hardware, and runs the whole program in supervisor state. For memory mapped I/O, such as used with the Motorola 68000, the task owning an I/O device can issue the I/O instructions directly in C — other processors, which have special I/O instructions, would require assembler subroutines. The actual I/O commands depend, of course, on the device, and might involve either programmed I/O or DMA. Synchronizing the task with the I/O device can be done busywait by reading status registers, either pre-busywait by checking that the device is ready before issuing an I/O command, or post-busywait after an I/O command is initiated by checking for the device to have completed before going on. There are situations where busywait synchronization is the preferred solution, because the busy state is guaranteed to clear in less time than would be required for a context switch to another task. However, in general it is unsatisfactory for three reasons: because in uselessly consuming processor cycles it prevents other tasks of the same or lower priority from running, because it is difficult to do with more than one device (especially input devices), and because it can readily lead to deadlocks. Generally, better practice is for the task to initiate an I/O command to the device, to synchronize by blocking while waiting for an interrupt to signal completion of the command, then to retrieve any results. (A pre-wait version, similar to pre-busywait, is occasionally appropriate instead.)

Harmony provides a primitive for awaiting an interrupt,

```
_Await_interrupt( interrupt_id, rply_msg);
```

The semantics of this primitive are something like those of the _Receive primitive. The task executing the primitive blocks until the interrupt identified by interrupt_id occurs. The rply_msg argument of the primitive is provided because experience has shown that many peripheral devices have volatile data which must be recorded at the instant the interrupt is first detected, for the data will be lost by the time that the waiting task can be activated. It is important to note that defining this primitive for most efficient implementation requires that at most one task can be waiting for a given interrupt, that a task can be waiting for only one interrupt, and, of course, that the waiting task must run on the processor physically connected to that particular interrupt.

Usually the task controlling a device will issue the I/O command, then call _Await_interrupt, and after the interrupt occurs and the primitive returns will access device registers to retrieve the results. It is important to notice that issuing the I/O command might cause the interrupt to happen immediately, before the task can call _Await_interrupt. Harmony treats interrupts as spurious when there is a protocol error, in that an interrupt occurs before there is a task waiting for it, as well as when noise or configuration errors result in interrupts occurring for which no handler has been provided. To ensure that this is not a problem, the task that will call _Await_interrupt for a particular interrupt must run at the Harmony priority corresponding to the hardware level at which that interrupt will occur, because then the priority scheduling rules ensure that the hardware will mask the interrupt until this task is blocked and the priority falls. Another common task structure is that one task calls _Await_interrupt to catch the interrupt when it happens, but a different task issues the I/O command whose completion will cause the interrupt. This is used, for instance, where several disk spindles are under a common controller. To resolve the critical section with respect to

controller registers, the same task issues all seek, read, and write commands, but because seeks, reads and writes on the different spindles can overlap, a separate notifier task is used for each spindle to catch the command completion interrupt. The task that issues the commands must reply to the notifier so that the latter can call _Await_interrupt before the interrupt happens, which as in the earlier case implies that the task calling _Await_interrupt must run at the Harmony priority corresponding to the hardware level at which the interrupt will occur. In this case, however, one must also ensure that the reply to the notifier is executed before the interrupt can happen, which requires either that the task that issues the I/O command must also run at the priority of the notifier, or that the reply be made before the I/O command is issued. (There have been some computers where the design of controller hardware actually forced tasks issuing the I/O command and awaiting the interrupt to run at different hardware priorities.)

It would be nice only to have to go down to this level of abstraction, but because Har - mony is an open system, indeed just because it is a configurable system, it is often necessary to descend further and implement the actual interrupt handler. Although doing so is not required, Harmony realizations typically use a two level interrupt handling scheme. The first level interrupt handler is invoked by the actual hardware interrupt, saves the registers on the stack of the interrupted task, and transfers to the proper second level interrupt handler for the particular processor interrupted. Having a separate first level is useful, for instance, if the hardware does not provide distinct interrupt vectors for each processor, and the first level interrupt handler must do that. (In this case the last step of state saving, saving the stack pointer in the task descriptor of the interrupted task, must be performed by the second level interrupt handler.) It is also useful if various members of a processor family have different registers that need to be saved, but a device, and hence the second level interrupt handler, might be used with any of these processors.

The second level interrupt handler must first identify the interrupt source. Although this may be available immediately as a consequence of vectoring, or may be available in some register, quite often it is not. Interrupts that the task level of abstraction would like to treat as distinct are often multiplexed for hardware reasons. For example, many UART (*universal asynchronous receiver transmitter*) chip designs only generate a single type of interrupt, even though software structures usually want different tasks to wait for input, output, and modem control events. Demultiplexing typically involves reading status registers, although if many devices have been connected to the same interrupt vector it may involve polling. Once the interrupt source has been identified, the second level interrupt handler must find whether there was a task waiting for that interrupt. The interrupt id used at the task level of abstraction is actually the byte offset in the vector _Int_table at which a pointer to the task descriptor of the waiting task, if any, is stored. (The interrupt id is also the byte offset into the vector _Dev_data_table at which is stored a pointer to any data which must be shared between the second level interrupt handler and the waiting task, such as readable copies of write-only device registers.) If the interrupt was spurious, that is if the entry of _Int_table was 0, the source of spurious interrupts is recorded in the external _Spurdev and the external _Spurcount is incremented. Volatile data associated with the interrupt must next be saved in the rply_msg provided by the waiting task, and then the interrupt must be cleared. If the interrupt was not spurious, the waiting task is now put on its ready queue and dispatched, else the interrupted task is reactivated.

The second level interrupt handler is typically code written in assembler, and because the stack of the interrupted task cannot be used and the task to dispatch has not yet been deter - mined, no stack is available. Interrupts are disabled throughout the second level interrupt handler. Note that when the waiting task is put on its ready queue, advantage can be taken of the fact that it will be the only task on that queue: had there been another, the priority of the

active task would have been at least that high, and the interrupt would not have been taken because it would have been masked by priority hardware.

Sometimes the above task level abstraction is not fast enough for the the arrival rate of the physical interrupts, and handling one interrupt might cause succeeding events to be missed. The problem might be that context switching itself is too slow, or it might be that too much immediate processing is required, or it might be that passing the processed data to another process by message passing is too slow. Often the real problem is that there is a clash in levels of abstraction: the task level of abstraction really wants to deal with high level events which occur at a rate that could be handled, but the hardware is causing physical interrupts for low level subevents which occur at a much higher rate — the problem could be finessed if the hard - ware device operated at the higher level. A classic example is the DDCMP communications protocol, which really is defined in terms of packets, but which is implemented with byte at a time interrupts through a standard UART. To solve such a problem in Harmony, we imple - ment such a higher level device outside of the Harmony abstraction, and use the concept of *fake interrupts* to activate the waiting tasks at the task level of abstraction. Implementing outside the Harmony abstraction could mean with a separate processor, or it could mean in software on the same processor, but with an interrupt at a hardware priority even above the Harmony disable. Obviously software outside the Harmony abstraction cannot make use of Harmony services, and preempting even disabled Harmony code sacrifices some of the strong things that can be proved about normal Harmony programs. In the DDCMP example, moving the packet between the UART and a buffer, including header and trailer processing and byte stuffing or stripping for transparency, can be done by a software automaton, running perhaps at unmaskable interrupt level. This ensures that the byte level handling will happen if the processor is at all fast enough to handle the communications line at that speed, and the task level interaction with this automaton is all in terms of packets, providing a buffer containing a packet to transmit or a buffer in which to receive a packet, and awaiting an interrupt to indicate completion of processing of the packet. Of course there is no such real interrupt, but the interrupt can be faked by jamming the waiting task directly on its ready queue, and forcing a redispatch by causing an interprocessor interrupt (which will be serviced immediately or, if the processor is disabled, as soon as it becomes enabled, or, if _Td_service is already in progress, when it completes). Jamming a task directly on its ready queue, regardless of what Harmony is doing, is safe if that task is the only task that can run at that priority, provided that the tail pointer of the ready queue is updated before the head pointer — the ready queue must have been empty, the only operation Harmony could have been doing with this ready queue is examining the head pointer to determine whether there is a task to dispatch, and so atomicity of writing the head pointer ensures it is safe.

The machinery described in this section is sufficient to provide direct control over any physical device. It should be recognized, however, that the task structure of Harmony facilitates creating *smart peripherals* where software implemented as a controlling task provides an interface closer to the level of device actually desired, instead of what the raw hardware provides. Not only do the messages serviced by this controlling task define a higher level device to client tasks, but if the controlling task runs on a different processor from that of the client, the latter sees the computational load offloaded too. Eventual migration of the *smart peripheral* task into specialized hardware is also made more convenient if that hardware uses the same interface. An example using a conventional computer peripheral would be a *file store device* implemented by a task controlling an ordinary disk, where the task also did space allocation, catalog management, etc., so that the operations on the smart device were to create and destroy linear address spaces, grow or shrink them as necessary, and read or write them — independent of physical block location, let alone cylinder boundaries, bad tracks, etc. An even higher level would be a *file system device* implemented by a task using the *file store device*, where this task provided a hierarchical file system with access control, concurrency control, etc. Another example, this time from robotics, would be a *vision system device*

implemented by a task controlling a TV camera and framegrabber, where the task responded to client tasks by indicating objects seen, together with their types, locations, and orientations.

Normally, a client task wanting to do I/O does it by exchanging messages with the server task which supports that I/O model for the desired device. The types of messages and the protocol of how they are used are what really define an I/O model. As discussed in the next section, an I/O server task must report for service to the directory task and must be prepared to receive open and close messages. What other messages it handles depend on the I/O model supported and on the device controlled. A simplified version of stream I/O, for instance, could be implemented by handling a read message to provide another buffer of input, a write message to output another buffer, a flush message to ensure that any buffers associated with the connection are flushed, as well as the open and close messages. The actual version of stream I/O provided with Harmony does not work this way, however, as the seek operation to move the current position in the stream has serious ramifications. Instead, the read and write requests are combined into a single advance window message, the model being that the stream should be thought of as the beginning part under control of the I/O server, concatenated with the window into the stream (the buffer) held by the client, concatenated with the ending part under control of the I/O server. When the user task is finished manipulating the current window (buffer), an advance window message is sent to the I/O server, returning the current window if it was modified, and requesting the next window if the connection is open for read.

Many server tasks may support the same I/O model. The server task may be the task directly controlling the hardware device or, as discussed above, a task implementing some higher level abstraction. The client task uses a symbolic name to establish, at run time, a connection with the appropriate server. This provides *device independence* within the class of servers supporting that I/O model.

## SERVERS AND CONNECTIONS

Harmony is a connection oriented system. What this means is that tasks that communicate frequently maintain state about each other, in particular about the effect of earlier communication. Connections are often discouraged in distributed systems, partly because they are assumed to be expensive, and partly because of the difficulty in coping with the inconsistencies that can affect this state when messages are lost or damaged. However, connections in Harmony are inexpensive, and under either the shared memory or thinwire model of computation damaged or lost messages are precluded. Most importantly, the effect of action requests in realtime systems is often to cause state changes in the real world (for example, the robot moves from its rest position, or the tracking radar locks on the selected target) and the program needs to reflect this real world state.

The function _Open was introduced in the discussion of stream I/O, in order to associate streams with servers supporting them. More generally, _Open is the mechanism for establishing a connection between a client task and a server. As stated earlier, _Open has two parameters, an open directive and a userid.

The open directive is not just a symbolic name for the server, but is more like a command line in other operating systems: it is a string beginning with a pathname of possibly several components and containing options for the connection to be established. The first component of the pathname, which can be at most 30 characters and includes a terminating ':', is the symbolic name for the server. Additional components, which are separated by '/', are interpreted by the particular server, for example a file system server might treat them as subdirectory names and the file name, or an automatic calling unit server might treat them as

parts of a telephone number. Options are position independent and are of three kinds: on toggles, off toggles, and value parameters. An on toggle is a '+' followed by a keyword, an off toggle is a '-' followed by a keyword, and a value parameter is a keyword and a value separated by an '='. A keyword is a sequence of characters delimited, for on toggles or off toggles by white space, and for value parameters by the '='. The value for value parameters is a sequence of characters delimited by white space. If a value containing white space is desired, the sequence of characters must be enclosed by quotation marks. (Note that if the open directive is a literal string, quotation marks enclosing a value inside the directive must be escaped.) The interpretation of on toggles and off toggles is left to the server, but the intended semantics are that a toggle has two values, on and off, and that there is possibly a default value which explicit appearance of the toggle overrides. The interpretation of value parameters is also left to the server, but the sequence of characters which form the value is made available as a string which the server may evaluate as the ASCII representation of a number, or may interpret as a symbolic name that can be looked up in an evaluate table to find a corresponding numeric value, or may parse further, or may simply use as text. Parsing open directives is supported by the library functions

```
svector = _Components( str );
svector = _On_toggles( str );
svector = _Off_toggles( str );
pvector = _Value_parameters( str );
number = _Evaluate( table, str );
check = _Valid_number( str );
_Freeparse( parse );
```

The userid argument to _Open is provided because some servers, for example a data base server, may want to enforce access controls on whether they will establish a requested connection. Having the userid be available directly to the application program, with the consequent risk of forgery, may seem unusual compared to the practice in multiuser timesharing systems, but when protection is needed in realtime applications it is usually is protection with respect to human operators interacting with an application program, not protection against the application program itself. Currently, requesting and validating the identity of an operator, and encoding this as a userid is completely left to the application program; no supporting functions are supplied. Most servers do not check the userid, consequently a value 0 is often used for this argument.

Calling _Open causes two messages containing the open directive to be sent. The first is to the directory task. The directory task matches the first component of the pathname against the list of symbolic names of registered servers, and returns the corresponding server id if a match is found, otherwise returning 0. If a server was found, the second message is sent, this time to the server. The server sets up the connection or can deny it, based on the rest of the pathname, on the userid, on what connections the server has already made, etc. The server then replies with a record containing a number identifying the connection, and the id of the task the client should communicate with for this connection (normally the server itself, but possibly an agent task instead), as well as the size of ucb_xtra block which this connection requires. (The ucb_xtra block is used, for instance, for buffers this connection requires on the client end.) Notice that by monitoring the success of this reply, the server can detect if the client task has been destroyed while the connection was being established, and thus can avoid leaving resources tied up in a connection which will not be used because the client no longer exists. Once a connection is established, messages from the client to the server must identify the connection, for a server may simultaneously be supporting multiple connections, even to the

same client task — the file system task might open several files for the same client task, for instance.

A connection is closed by the client task sending a close message to the server; this will be done automatically for any connections still open when a client task is destroyed. This is the third major aspect to connections (the other two being runtime symbolic binding and textual initialization options). In many situations, it is important for one task to know when another task dies, in order to perform wrapup, release resources, spawn replacement tasks, etc. This can be done without the connection mechanism, but only by creating vulture tasks that do a receive specific on the task to be monitored without ever being sent to by it, relying on the fact that the message passing primitives fail when the correspondent dies. The connection mecha - nism is much cheaper and more explicit.

The directory task initially has no built-in knowledge of which servers are available, so any server must send a REPORT_FOR_SERVICE message to the directory task reporting the symbolic name by which it is known. A server may also be known by secondary names. The reporting is conveniently accomplished by calling

```
_Report_for_service( name, msg_type );
```

where name is a string and msg_type is one of REPORT_SECONDARY_NAME or REPORT_FOR_SERVICE. Any secondary names must be reported before the report for service. It is possible for a different task to register at a later time under the same name, thus temporarily or permanently taking on the responsibility of providing the services identified by that name.

To facilitate a server keeping track of those connections it has open with clients, routines are provided to allocate a connection table, allocate an entry in a connection table, lookup a connection table entry, and free a connection table entry

```
table      = _Alloc_connection_table( init_num_entries,
                 grow_num_entries, max_num_entries,
                 data_blk_size );
scb        = _Get_connection( table, client, new_connection,
                 con_data_blk );
scb        = _Lookup_connection( table, client, connection );
connection = _Free_connection( table, connection );
```

where init_num_entries is the number of entries initially (the table is grown later as needed), grow_num_entries is the amount by which the table is to be grown each time it grows, max_num_entries is the maximum size beyond which the table will not be grown (except that a 0 value indicates no limit), data_blk_size is the size needed for the server connection block for this server, table is a pointer to the allocated structure, client is the id of the task with which the connection is being made, new_connection is a pointer to where the new connection number should be returned, con_data_blk, if supplied, is a pointer to the connection data block, scb is a pointer to the server connection block, connection is the number of the connection to be looked up or freed, and _Free_connection returns its connection argument as its function value or 0 if it fails.

The application program is responsible for creating whatever servers it will require. This is best done at the start of function main(), in the first user task. Care must be taken to allow for the critical race where the server must report for service to the directory task before the first

client task can open a connection with it. This is conveniently done by using the _Server_create function

```
id = _Server_create( global_index, init_list );
```

which creates an instantiation of the specified task template, if init_list is nonzero supplies the initialization records on init_list, one at a time, as reply messages to the newly created server task as it requests them, and returns the id of the server task after the task has sent a message confirming that reporting is complete. _Server_create returns 0 upon failure to create and initialize the server.

The *scatter/gather* aspect of init_list is very important in designing generic servers. A server is generic when several similar servers can be combined as instantiations of the same one by supplying the differences through initialization records. For instance, instead of having a different server for each terminal on a system, each with device addresses and characteristics hard coded into the functions the servers execute, the use of a generic server not only saves code space but also greatly facilitates reading the device assumptions a program makes and changing them. For reading and managing such configurations, a static description in a compile time initialized record structure is greatly preferable to attempting to combine all the device characteristics in a single record, or to building the configuration record at run time. Configuration record structures maintained in files and accessed as needed at run time also benefit from exploiting scatter/gather for modularity. The formats of the records in init_list are server dependent, but each must begin with the following struct

```
struct INIT_REC
 {
  struct STD_RQST   STANDARD;
  struct INIT_REC   *IR_NEXT;
 }
```

where STD_RQST contains two short unsigned fields: MSG_SIZE and MSG_TYPE, MSG_TYPE in this case being used to distinguish the various types of initialization records for a particular server. A server requiring initialization records should continue to request subsequent records until IR_NEXT is 0.

The design and use of servers is a sufficiently important topic that although servers have been referred to extensively in the foregoing discussion, it is worth collecting together some of the issues. The first thing to recognize is that a server is analogous to a library subroutine. Whereas a private subroutine that is called from only a few well-known places can rely on undocumented aspects of the context in which it is called, a library subroutine must have clean, well-defined interfaces so that it will be useful to call from code not even thought of at the time the library subroutine was written. Like a library subroutine, the server task must provide a generally useful service, and the need for clean, well-defined interfaces applies to servers as well. What distinguishes a server from a library subroutine is that the server often must maintain state across service transactions, perhaps just across transactions for the same client, or perhaps even across clients. This state is inconvenient to represent in data carried by the clients, but can be conveniently represented in data local to the server task. Because the server may maintain state for several clients at any time, it must have a way to distinguish them. Connection ids which the server can issue to clients solve this, and also allow a client to have more than one open connection to the same server. Because the client will not, in general, have created the server, it cannot be expected to have inherited the task id of the server from somewhere, and so must identify the server by symbolic name. As Harmony tasks are

identified by system produced task ids (there are no *well-known names* in networking jargon), an explicit lookup of the symbolic name is required. This facilitates context dependent or time dependent binding of symbolic name to task id. Given that the mechanism Harmony provides for such binding is connections, a server must support OPEN and CLOSE messages, must register with the _Directory task, and must provide connection numbers. Servers often have some parameters that must be set before normal client transactions can be handled, so the toggles and value parameters in the OPEN directive can be used to ensure these parameters are set before use. A server often needs to know if a client dies, perhaps while the service transaction is in service or perhaps between service transactions, so the system supplied CLOSE messages on connections are important. To reduce the overhead of having a send–receive–reply on every service request, a server may require the client to provide a buffer in the client's space to provide some caching. This buffer would be in the UCB_XTRA, whose use is dependent on the particular server and on any library functions used by the client to interact with that server. It should be noted that because a server accepts one request at a time, it serializes access to any shared resource which it controls and thus can solve critical races and mutual exclusion problems. It need not process only one request at a time, that is, it need not complete processing one request before accepting another. For simple nonsharable but serially reusable resources, having the server perform operations on the resource on behalf of the client may be adequate. For complex sharable resources, such as an application data structure, the server may allow concurrent access by workers or by the clients themselves, while enforcing consistency conditions such as atomic updates, two phase commit, or access only to disjoint substructures.

Several examples of servers are included in Harmony, including a calendar clock server, a server to provide explicit scheduling, servers to support interactive terminals and to act as virtual terminals to remote computers, a TCP/IP Ethernet server, and a file device server and file system server.

## USER'S VIEW OF A PROGRAM

Recapitulating and extending what was said above, a user sees a program as a set of processor images, one for each processor in the hardware configuration. A processor image is obtained by linking three fundamental user supplied externals with whatever user supplied externals and functions are brought in, directly or indirectly, by the references in the three fundamental externals, as well as with the Harmony kernel and support libraries. (In a ROM system, of course, the Harmony kernel and support libraries linked to are simply ROM entry points.) The preferred method of program development is thus that all user code and data, other than the three fundamental externals for each processor, is maintained in relocatable object code libraries.

The three fundamental externals for a processor image are

```
unsigned _Pnumber =
```

which must be initialized with a value specifying which processor this image is for;

```
struct TASK_TEMPLATE _Template_list[] =
```

which must be initialized as a vector of all task templates for tasks that can be created on this processor; and

```
struct INT_PAIR _Interrupt_list[] =
```

which must be initialized as a vector of all interrupt handlers required for this processor (excluding the interprocessor interrupt, which is always included). The two vectors must be terminated by a null task template and a null int pair, respectively, i.e., in which all fields are 0. The _INT_PAIR entries are the following struct

```
struct INT_PAIR
 {
   unsigned INT_PHYSICAL_ID;
   void     (*INT_HANDLER)();
 }
```

where the field INT_PHYSICAL_ID is a 32 bit unsigned that identifies the physical interrupt in whatever way is appropriate to that machine (interrupt priority level, bus vector, interrupt vector address, etc.) and INT_HANDLER is the second level interrupt handler code to be executed when that interrupt is taken.

On processor 0, three special task templates must be included in _Template_list. A task template with GLOBAL_INDEX of 1 must be provided as the first user task, which the system starts and for which the root function is normally the user function main(). This is the start of the user's program. A task template with GLOBAL_INDEX of 2 must also be provided as the directory task, for which the root function would normally be _Directory, but which might conceivably have other than the default priority of 5. Finally, a task template with GLOBAL_INDEX of 3 must also be provided as the gossip task, for which the root function would normally be _Gossip, but which might conceivably have other than the default priority of 5.

In choosing how to structure the source code, it is wise to remember that all compilation source should have an include file sys.h, in order to have necessary struct and manifest constant declarations. It is also good practice to include a file giving all template index assignments to be used throughout the whole program. Remember to include the templates for all servers that will be used, as well as for the worker tasks they create. When choosing identifiers, note that Harmony uses the convention that all Harmony-defined objects, both internal and visible, start with an underscore in order that accidental name conflicts can be avoided and system use can be recognized. Another convention is that struct tags, field names, and manifest constants are all defined by upper case only names, and that externals and functions have names for which the first alphabetic is upper case and there is at least one lower case alphabetic.

Linker parameters are different for processor images for different processors, which is most conveniently handled by working with libraries, making linking an explicit step in program building, and using a specific command file for linking processor images for each processor. On the Chorus machine each processor is defined with RAM starting on a megabyte boundary. The processor image is linked with that as base. The first routine loaded into the image is forced to be _Pre_Setup which must be at a known address because control transfers there to start execution. This routine loads four addresses in the application program which the linker determines and the kernel needs to know (the locations of the three fundamental externals and the first location available for the storage pool) into registers and then transfers to _Setup. The latter, which can be anywhere including in ROM, sets up the C environment, then calls the Harmony kernel to build and initialize Harmony. Processor 0 initializes its Harmony kernel first, then opens the multiprocessor gates so the other processors can initialize their Harmony kernels. (A gate is a location used for synchronization.) In a

downloading situation, on reset all processors other than processor 0 close their multiprocessor gate, then enter a busywait loop waiting for their multiprocessor gate to open, and processor 0 executes whatever code is necessary to accept a download of code first for the other processors, and then for itself. The end of downloading causes it to execute _Pre_Setup and initialize itself. Processor 0 must then open the multiprocessor gates for the other processors, then poll the gates to determine when all the other processors have completed initialization before it can start the first user task. The other processors initialize when their gates open.

## SUPPORT TOOLS

Harmony currently provides several support tools, including listing, examine, bound, and debug. These are all at a very primitive stage of development. The first three, i.e., listing, examine, and bound, all run in the development environment rather than in Harmony itself; examine and bound are intimately involved with the exact form of a stored executable image and with the generated code from the current compiler. Consequently these tools may need considerable conversion to be used outside the environment in which Harmony was initially developed. Nevertheless, they do serve as prototypes for tools that are useful.

### listing

The style of programming encouraged by Harmony consists of many small functions, each stored in a separate file, and many header files, each defining structs and macros associated with a different abstraction. Compilations are done by compiling inclusion files that reference the appropriate source. A readable listing needs to be arranged as a book, with cover, table of contents, numbered pages with headings, and a concordance or index at the end. This tool, given an inclusion file, produces this.

### examine

It is sometimes necessary to examine a bound executable image, to determine overall characteristics such as the size of the text or data segments, to look up external symbols, to disassemble instructions or display initialized data, or even to patch erroneous values. This tool, given a .exe file produced by Whitesmiths' linker or a .out file produced by the Unix linker, does this.

### bound

Bounding the stack required for a given task involves discovering the call graph of procedures called from the root function, or called by procedures called from the root function, etc. The stackframe size required for each of these procedures must be determined. All possible nesting lists can then be simulated to determine the maximum stack size required. Two things force this to be an interactive, rather than a wholly automatic, process. First, sometimes there are indirect procedure calls, i.e., calls to procedures which are formal arguments of the calling procedure or which are values returned by calls to other procedures or which are values of procedure-type variables. These cannot be known without running the program, so the possible procedures must be supplied interactively. Second, the call graph may involve a cycle, implying possible recursion, and while the stack requirement per cycle can be made available, the number of levels of recursion to support must be supplied interactively. This tool, given a .exe file or a .out file, allows the user to determine stack sizes for several root functions in that file.

### debug

The question of how to debug a multitask program is still very much unanswered and may even require hardware support. For now, this very simple debugger is all that is available for Harmony. Breakpoints can be planted at run time or can be compiled into the source as calls to the function _Breakpoint. Values to be inspected must be identified by absolute address, and values are only available in decimal or hexadecimal. On the other hand, the debugger can print certain known system data structures, such as task descriptors, and works with interrupts enabled and with breakpoints on code in processors other than the one with the terminal. The debugger also supports *use bits*, a debugging technique for recording whether an event occurred that is sufficiently efficient to be left in a production program. Considerable evolution of the debugger can still be expected. The current version requires a stream I/O server "DEFAULT:" to be available.

## PERFORMANCE

It is difficult for a portable system to make quantitative statements about performance, since measured values depend upon tuning of the kernel code, which compiler is chosen, the microprocessor instruction set, the clock speed, the memory used, other aspects of the proces - sor board design, etc. Nevertheless, qualitative statements about performance are important, to make sensible design tradeoffs in application programs. The measurements described here were made on a system with kernel code written for clarity rather than tuned to the target, using a processor that was a 10 MHz 68000 with no wait states, Harmony having been compiled by Whitesmiths' C compiler.

The first important characteristic is response time to an interrupt, i.e., delay from the moment of interrupt until the _Await_interrupt of the waiting task returns. This is measured as 0.100 ms. An indication of how good this really is can be seen from the fact that the fastest assembly language interrupt handler, operating outside the tasking abstraction but saving the state of the interrupted task and sorting out which processor's interrupt vector to use, takes 0.040 ms.

The second important characteristic is the send–receive–reply cycle time. When both tasks are running in the same, otherwise empty, processor, this is measured as 1.15 ms. When the two tasks are running in different, otherwise empty, processors, this is measured as 1.10 ms, the reduction coming about because of parallelism in execution of the primitives on the two processors. Also important are how much processor time is required on the sending processor and how much on the receiving processor. These are measured as 0.460 ms and 0.720 ms, respectively.

An important aspect of the message passing implementation is that it is distributed. Only the processor or processors that the communication tasks are running on participate in the protocol: there is no central *switching* processor involved. The actual bus cycles used on the shared bus are so few that contention for these will not be a bottleneck. Message passing between tasks on the same processor is treated uniformly with message passing between tasks on different processors and not by special code.

The measured times given above are quite commensurate with the best task context switching times achieved by other systems on similar hardware, even uniprocessor systems.

In some situations, however, they are not fast enough. This problem has been recognized as being due to inadequate hardware support for operating system primitives, and one solution

often proposed is to provide an operating system coprocessor for each processor or to provide microcode support for the primitives. Although this has not yet been tried for Harmony, it has been very effective for similar systems, providing more than an order-of-magnitude speedup. Consequently, the message passing protocol for Harmony is implemented by a single identifiable piece of code that should be moved to a coprocessor or to microcode if that is available.

## ACKNOWLEDGEMENT

Harmony, of course, is built upon concepts proved out in earlier operating systems and languages, as well as new ideas evolved for it. Particular mention should be made of Thoth (2, 3), where this particular process abstraction originated, and BCPL, where the I/O abstraction originated and whose global vector was the inspiration for the global template list. (The I/O abstraction is also commonly associated with Unix, which copied it from BCPL).

## REFERENCES

1. W.M. Gentleman. "If only the hardware... (A software designer's lament)." Proceedings of the IEEE International Workshop on Computer Systems Organization, New Orleans, LA. March 29–31, 1983. pp. 88–95.

2. W.M. Gentleman. "Message passing between sequential processes: the reply primitive and the administrator concept." Software Pract. Exper. 11(5): 435–66; 1981.

3. D.R. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager. "Thoth, a portable real-time operating system." Commun. ACM, 22(2): 105–115; 1979.

4. J.D. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Brueckner, O. Roubine, and B.A. Wichmann. "Rationale for the Design of the Ada programming language." SIGPLAN Notices, 14, 6. Part B, June 1979.

# APPENDIX A. A SAMPLE MULTITASK PROGRAM

The following program is a simple multitask program which illustrates many of the ideas and primitives discussed in this manual. In this Appendix, the program is run on only processor 0. The function main() is the root function of the first user task. It creates a server for a terminal attached to a serial port and opens this for input and output. It then creates a child task and goes into a loop, sending to the child task and printing the response and accompanying text to the selected terminal.

This program also illustrates the source management strategy discussed in Appendix E. The example is from [harmony.example.srtest] in the distribution tree, and the inclusion file shown is for the version of the program to run on a Chorus machine with Omnibyte OB68K1A boards with the 68000 processor, the program to be compiled on a VAX under VMS, using Whitesmiths' compiler.

**What is actually compiled and linked, i.e. the inclusion file**
**[.inc.ch00wvms]srtest.c**

```
#include "[----.sys.src]sys.h"
#include "[----.sys.src.connect]connect.h"
#include "[----.sys.src.servers.tty]ttyinit.h"
#include "[----.sys.src.servers.tty.mc6850.whtsmthsa]serialint.h"
#include "[--.src.chorus]templates.h"
#include "[--.src.chorus]srtest.c"
#include "[--.src.chorus.whtsmthsc]ttyinit.c"
#include "[--.src]child.c"
#include "[--.src]main.c"
```

**The following header files are not shown**
**[----.sys.src]sys.h**
**[----.sys.src.connect]connect.h**
**[----.sys.src.servers.tty]ttyinit.h**
**[----.sys.src.servers.tty.mc6850.whtsmthsa]serialint.h**

**The files containing code are:**

**[--.src.chorus]templates.h**

```
/*
 * Global indices for task templates:
 * Note that indices 1-3 are reserved for main, _Gossip, and _Directory.
 */

#define TTI             4
#define TTO             5
#define TTY             6
#define CHILD           10
```

**[--.src.chorus]srtest.c**

```
extern main();
extern _Directory();
extern _Gossip();
extern Child();
extern _Tty_server();
extern _Tti();
extern _Tto();

extern _Ptm_int();
extern _Serial_int();

uint_32 _Pnumber = 0;

struct TASK_TEMPLATE _Template_list[] =
 {
    { MAIN, main, 600, 7, 0 },
    { DIRECTORY, _Directory, 202, 7, 0 },
    { GOSSIP, _Gossip, 482, 5, 0 },
    { CHILD, Child, 400, 6, 0 },
    { TTY, _Tty_server, 590, 5, 0 },
    { TTI, _Tti, 104, 0, 0 },
    { TTO, _Tto, 134, 0, 0 },
    { 0, 0, 0, 0, 0 }
 };

struct INT_PAIR _Interrupt_list[] =
 {
   { 4, _Ptm_int },
   { 5, _Serial_int },
   { 0, 0 }
 };
```

**[--.src.chorus.whtsmthsc]ttyinit.c**

```
/*
 * This file contains initialization records used for P0_TTY0, which is
 * configured as the port on ACIA 0 on processor 0 of an Omnibyte OB68K1A.
 * This port also has the  secondary name DEFAULT.
 * Modem control and other interrupts are not supported.
 */

/* declarations to satisfy compiler re forward references */
extern struct TTYI_ALTNAME_INIT_REC Default_init;

struct TTYI_PORT_INIT_REC P0_tty0_init =
 {
   sizeof( struct TTYI_PORT_INIT_REC ),        /* MSG_SIZE */
```

```
  TTYIT_PORT_INIT,                    /* MSG_TYPE */
  (struct INIT_REC *)&Default_init,   /* next initialization record */
  "P0_TTY0:",                         /* tty server name */
  RECV_0ACIA,                         /* tti device code */
  TTI,                                /* tti template index */
  XMIT_0ACIA,                         /* tto device code */
  TTO,                                /* tto template index */
  ETC_0ACIA,                          /* ttetc device code not used */
  0,                                  /* ttetc template index not used */
  (char *)TTY0_ADDR,                  /* I/O address of tty port */
  0                                   /* baud rate - not used on Chorus */
};

struct TTYI_ALTNAME_INIT_REC Default_init =
{
  sizeof( struct TTYI_ALTNAME_INIT_REC ),  /* MSG_SIZE */
  TTYIT_ALTNAME_INIT,                      /* MSG_TYPE */
  0,                                       /* no more initialization records */
  "DEFAULT:",                              /* tty server name */
};
```

**[--.src]main.c**

```
main()
{
  extern struct TTYI_PORT_INIT_REC P0_tty0_init;

  uint_32 n, limit;
  uint_32 child;
  struct STD_RQST request;
  struct STD_RPLY reply;

  _Server_create( TTY, (struct INIT_REC *)&P0_tty0_init );
  _Selectinput( _Open( "P0_TTY0: +r", 0 ) );
  _Selectoutput( _Open( "P0_TTY0: +w", 0 ) );

  _Putstr( "parent creates child\n" );
  _Flush();
  child = _Create( CHILD );

  limit = 100;
  for(;;)
  {
    for( n=0; n<limit; ++n )
    {
      request.MSG_SIZE = sizeof( request );
      reply.MSG_SIZE = sizeof( reply );
      _Send( (char *)&request, (char *)&reply, child );
```

```
        _Putstr( "child replied " );
        _Putdec( reply.RESULT );
        _Putstr( "\n" );
      }
      _Putstr( "How many more message exchanges do you want? " );
      _Flush();
      limit = _Getnum();
    }
  };
```

**[--.src]child.c**

```
Child()
 {
   uint_32 counter, requestor;
   struct STD_RQST request;
   struct STD_RPLY reply;

   counter = 0;

   for(;;)
    {
      request.MSG_SIZE = sizeof( request );
      requestor = _Receive( (char *)&request, 0 );

      reply.MSG_SIZE = sizeof( reply );
      reply.RESULT = counter++;
      _Reply( (char *)&reply, requestor );
    }
 };
```

# APPENDIX B. A SAMPLE MULTITASK MULTIPROCESSOR PROGRAM

This program is the same as the program of Appendix A, but the child task has been moved to run on processor 1. Note that the only difference in the program is that the task template for the child task has been moved from the _Template_list for processor 0 to the _Template_list for processor 1. Had the ttyinit record, main(), and Child() been compiled into a library of relocatable object, the inclusion files defining the processor images could have been even simpler, essentially just referring to the files containing the three fundamental externals.

**What is actually compiled and linked for processor 0, i.e. the inclusion file [.inc.ch00wvms]srtest0.c**

```
#include "[----.sys.src]sys.h"
#include "[----.sys.src.connect]connect.h"
#include "[----.sys.src.servers.tty]ttyinit.h"
#include "[----.sys.src.servers.tty.mc6850.whtsmthsa]serialint.h"
#include "[--.src.chorus]templates.h"
#include "[--.src.chorus]srtest0.c"
#include "[--.src.chorus.whtsmthsc]ttyinit.c"
#include "[--.src]main.c"
```

**What is actually compiled and linked for processor 1, i.e. the inclusion file [.inc.ch00wvms]srtest1.c**

```
#include "[----.sys.src]sys.h"
#include "[--.src.chorus]templates.h"
#include "[--.src.chorus]srtest1.c"
#include "[--.src]child.c"
```

**Files containing additional code not shown in Appendix A: [--.src.chorus]srtest0.c**

```
extern main();
extern _Directory();
extern _Gossip();
extern _Tty_server();
extern _Tti();
extern _Tto();

extern _Ptm_int();
extern _Serial_int();

uint_32 _Pnumber = 0;

struct TASK_TEMPLATE _Template_list[] =
 {
    { MAIN, main, 600, 7, 0 },
    { DIRECTORY, _Directory, 202, 7, 0 },
```

```
    { GOSSIP, _Gossip, 482, 5, 0 },
    { TTY, _Tty_server, 590, 5, 0 },
    { TTI, _Tti, 104, 0, 0 },
    { TTO, _Tto, 134, 0, 0 },
    { 0, 0, 0, 0, 0 }
 };

struct INT_PAIR _Interrupt_list[] =
 {
   { 4, _Ptm_int },
   { 5, _Serial_int },
   { 0, 0 }
 };
```

**[--.src.chorus]srtest1.c**

```
extern Child();

extern _Ptm_int();
extern _Serial_int();

uint_32 _Pnumber = 1;

struct TASK_TEMPLATE _Template_list[] =
 {
    { CHILD, Child, 400, 6, 0 },
    { 0, 0, 0, 0, 0 }
 };

struct INT_PAIR _Interrupt_list[] =
 {
   { 4, _Ptm_int },
   { 5, _Serial_int },
   { 0, 0 }
 };
```

# APPENDIX C. DECLARATIONS

The following typedefs allow us to minimize portability problems due to the various C compilers (even for the same processor) not agreeing on the sizes of int and short int.

```
typedef   unsigned char   uchar;      /* unsigned characters */
typedef            short  int_16;     /* 16-bit signed integers */
typedef   unsigned short  uint_16;    /* 16-bit unsigned integers */
typedef            long   int_32;     /* 32-bit signed integers */
typedef   unsigned long   uint_32;    /* 32-bit unsigned integers */
```

The headings of user called procedures follow. Some of these are interfaces to servers, and hence are only relevant when those servers are used.

**Memory management**

```
char *_Getvec( size )
   uint_32 size;

_Freevec( block )
   char *block;

_Trimvec( block, size )
   char *block;
   uint_32 size;

uint_32 _Sizeof( block )
   char *block;

_Tune_Getvec( size )
   uint_32 size;
```

**Task creation and destruction**

```
uint_32 _Create( task_index )
   uint_32 task_index;

_Destroy( id )
   uint_32 id;

void _Suicide()

uint_32 _Father_id()

uint_32 _My_id()
```

**Message passing**

```
uint_32 _Send( rqst_msg, rply_msg, id )
   char *rqst_msg, *rply_msg;
   uint_32 id;
```

```
uint_32 _Receive( rqst_msg, id )
   char *rqst_msg;
   uint_32 id;

uint_32 _Try_receive( rqst_msg, id )
   char *rqst_msg;
   uint_32 id;

uint_32 _Reply( rply_msg, id )
   char *rply_msg;
   uint_32 id;
```

**Interrupts**

```
_Await_interrupt( interrupt_id, rply_msg )
   uint_16 interrupt_id;
   char *rply_msg;

_Disable()

_Enable()
```

**Error handling**

```
_Abort( s )
   char *s;

int_32 _Stackoverflow()

uint_32 _Set_task_error_code( error_code )
   uint_32 error_code;

uint_32 _Task_error_code()
```

**Debugging**

```
_Log_gossip( s )
   char *s;

_Breakpoint( s )
   char *s;
_Debug( dbg_state )
   struct DBG_STATE *dbg_state;

uint_32 _Get_use_bit_vector( dc_ucb, pnumber, num_bits )
   struct UCB *dc_ucb;
   uint_32 pnumber, num_bits;
```

```
_Put_use_bit_vector( dc_ucb, pnumber )
   struct UCB *dc_ucb;
   uint_32 pnumber;

uint_32 _Rd_use_bit_vector( dc_ucb, pnumber, buf, buf_size )
   struct UCB *dc_ucb;
   uint_32 pnumber;
   uchar *buf;
   uint_32 buf_size;

boolean _Clr_use_bit_vector( dc_ucb, pnumber )
   struct UCB *dc_ucb;
   uint_32 pnumber;
```

**Implementing alternative debugging environments**

```
boolean _Set_breakpoint( dc_ucb, addr, task_id, bp )
   struct UCB *dc_ucb;
   uint_32 addr, task_id;
   struct USR_BP_RECORD *bp;

boolean _Find_breakpoint( dc_ucb, bp_number, bp_next, bp )
   struct UCB *dc_ucb;
   uint_16 bp_number;
   boolean bp_next;
   struct USR_BP_RECORD *bp;

boolean _Hit_breakpoint( dc_ucb, addr, task_id, bp )
   struct UCB *dc_ucb;
   uint_32 addr, task_id;
   struct USR_BP_RECORD *bp;

boolean _Clr_breakpoint( dc_ucb, bp_number, bp )
   struct UCB *dc_ucb;
   uint_16 bp_number;
   struct USR_BP_RECORD *bp;

_List_breakpoints( dc_ucb )
   struct UCB *dc_ucb;
```

**Using connections**

```
struct UCB *_Open( name, user_id )
   char *name;
   uint_16 user_id;

_Close( ucb )
   struct UCB *ucb;
```

```
uint_32 _Server_create( task_index, init_list )
   uint_32 task_index;
   struct INIT_REC *init_list;
```

**Stream I/O**

```
struct UCB *_Selectinput( ucb )
   struct UCB *ucb;

struct UCB *_Selectoutput( ucb )
   struct UCB *ucb;

_Seek( ucb, position, relative )
   struct UCB *ucb;
   int_32 position;
   uint_32 relative;

char _Get()

char _Put( byte )
   char byte;

_Flush()

_Unget()

int_32 _Getnum()

uint_32 _Getstr( s, n )
   char *s;
   uint_32 n;

_Putdec( n )
   int_32 n;

_Puthex( n )
   int_32 n;

_Putstr( s )
   char *s;

_Printf( fmt, x )
   char *fmt;
   int_32  x;
```

**Utilities**

```
_Copy( src, dest, num_bytes )
   char *src, *dest;
   uint_32 num_bytes;
```

```
_Zero( ptr, num_bytes )
   char *ptr;
   uint_32 num_bytes;

_Sprintf( str, fmt, x )
   char *str, *fmt;
   int_32  x;

int_32 _Sgetnum( s )
   register char *s;

int_16 _Str_compare( s1, s2 )
   register char *s1, *s2;

uint_32 _Str_length( s )
   char *s;

uint_16 _Swap_16( in_16 )
   uint_16 in_16;

uint_32 _Swap_32( in_32 )
   uint_32 in_32;
```

**Parsing utilities**

```
char **_Components( s )
   char *s;

char **_Off_toggles( s )
   char *s;

char **_On_toggles( s )
   char *s;

struct VALUE_PARAMETER *_Value_parameters( s )
   char *s;

boolean _Valid_number( s )
   register char *s;

int_32 _Evaluate( table, s )
   struct EVALUATE_ENTRY table[];
   char *s;

_Freeparse( parse )
   char **parse;
```

**Implementing servers**

```
struct CON_TABLE *_Alloc_connection_table( init_num_entries,
                                           grow_num_entries, max_num_entries,
                                           data_blk_size )
   uint_32 init_num_entries, grow_num_entries, max_num_entries;
   uint_32 data_blk_size;

uint_32 _Free_connection( table, connection )
   struct CON_TABLE *table;
   uint_32 connection;

char *_Get_connection( table, client, new_connection, con_data_blk )
   struct CON_TABLE *table;
   uint_32 client, *new_connection;
   char *con_data_blk;

char *_Lookup_connection( table, client, connection )
   struct CON_TABLE *table;
   uint_32 client, connection;

uint_32 _Report_for_service( name, msg_type )
   char *name;
   uint_16 msg_type;
```

**Interface to clock server**

```
uint_16 _Getmonth( clock_ucb )
   struct UCB *clock_ucb;

uint_16 _Setmonth( clock_ucb, month )
   struct UCB *clock_ucb;
   uint_16 month;
uint_32 _Gettime( clock_ucb )
   struct UCB *clock_ucb;

uint_32 _Settime( clock_ucb, millisec )
   struct UCB *clock_ucb;
   uint_32 millisec;

uint_32 _Delay( clock_ucb, millisec )
   struct UCB *clock_ucb;
   uint_32 millisec;

uint_32 _Sleep( clock_ucb, millisec )
   struct UCB *clock_ucb;
   uint_32 millisec;
```

```
uint_32 _Wakeup( clock_ucb, wake_id )
   struct UCB *clock_ucb;
   uint_32 wake_id;

uint_32 _Time_from_vec( vec )
   uint_16 vec[5];

_Time_to_vec( time, vec )
   uint_32 time;
   uint_16 vec[5];
```

**Interface to file system server**

```
char *_Copy_file( from_name, to_name, user_id )
   char *from_name, *to_name;
   uint_16 user_id;

char *_Copy_tree( fsys_ucb, from_root, to_root, user_id )
   struct UCB *fsys_ucb;
   char *from_root, *to_root;
   uint_16 user_id;

char *_Get_current_node( fsys_ucb, pathname )
   struct UCB *fsys_ucb;
   char *pathname;

char *_Get_access_perms( fsys_ucb, pathname, owner, o_perms, w_perms )
   struct UCB *fsys_ucb;
   char *pathname;
   uint_16 *owner;
   uchar *o_perms;
   uchar *w_perms;

char *_Make_node( fsys_ucb, pathname, o_perms, w_perms  )
   struct UCB *fsys_ucb;
   char *pathname;
   uchar o_perms, w_perms;

struct SUBSTRUCTURE_ENTRY *_Next_substructure_entry( entry )
   struct SUBSTRUCTURE_ENTRY *entry;

struct RP_U_BITMAP_READ *_Read_bitmap( fsys_ucb, ldrv_no, bm_length,
                                       bm_buf )
   struct UCB *fsys_ucb;
   uint_32 ldrv_no, bm_length;
   struct RP_U_BITMAP_READ *bm_buf;
```

```
struct SD *_Read_space_descriptor( fsys_ucb, pathname, sd )
   struct UCB *fsys_ucb;
   char *pathname;
   struct SD *sd;

struct RP_U_SUBSTRUCTURE_READ *_Read_substructure( fsys_ucb, pathname,
                                                   sb_length, substr_buf )
   struct UCB *fsys_ucb;
   char *pathname;
   uint_32 sb_length;
   struct RP_U_SUBSTRUCTURE_READ *substr_buf;

char *_Rename_node( fsys_ucb, pathname, new_node_name )
   struct UCB *fsys_ucb;
   char *pathname, *new_node_name;

char *_Rm_node( fsys_ucb, pathname )
   struct UCB *fsys_ucb;
   char *pathname;

char *_Rm_tree( fsys_ucb, root )
   struct UCB *fsys_ucb;
   char *root;

char *_Set_current_node( fsys_ucb, pathname )
   struct UCB *fsys_ucb;
   char *pathname;

char *_Set_access_perms( fsys_ucb, pathname, change_owner, new_owner,
                         o_on, o_off, w_on, w_off, owner, o_perms, w_perms )
   struct UCB *fsys_ucb;
   char *pathname, change_owner;
   uint_16 new_owner, *owner;
   uchar o_on, o_off, w_on, w_off, *o_perms, *w_perms;
char *_Shrink_file( pathname, user_id )
   char *pathname;
   uint_16 user_id;

char *_Walk_tree( fsys_ucb, pathname, walk_data, traversal )
   struct UCB *fsys_ucb;
   char *pathname;
   char *walk_data;
   uint_32 traversal;
```

**Interface to display file system information**

```
char *_File_size( pathname, user_id )
   char *pathname;
   uint_16 user_id;
```

```
uint_32 _Fsys_space( fsys_ucb, init_list )
   struct UCB *fsys_ucb;
   struct INIT_REC *init_list;

char *_List_file( pathname, user_id, break_pages )
   char *pathname;
   uint_16 user_id;
   char break_pages;

char *_List_nodes( fsys_ucb, pathname, check_substr )
   struct UCB *fsys_ucb;
   char *pathname, check_substr;

char *_Put_tree( fsys_ucb, root )
   struct UCB *fsys_ucb;
   char *root;

char *_Stat_node( fsys_ucb, pathname, user_id )
   struct UCB *fsys_ucb;
   char *pathname;
   uint_16 user_id;
```

**Interface to file device server**

```
uint_32 _Get_file_size( file_ucb, bytes_allocated, blks_allocated )
   struct UCB *file_ucb;
   uint_32 *bytes_allocated, *blks_allocated;

uint_32 _Set_file_size( file_ucb, bytes_requested, blks_allocated )
   struct UCB *file_ucb;
   uint_32 bytes_requested, *blks_allocated;
```

**Interface to file device formatting server**

```
uint_32 _Fmt_disk( fdf_ucb, config, pattern, interleave_factor )
   struct UCB *fdf_ucb;
   struct FD_PDRV_INIT_REC *config;
   uint_32 pattern;
   uint_16 interleave_factor;

uint_32 _Fmt_track( fdf_ucb, pdrv_no, cylinder, head, fmt_type, pattern,
                    interleave_factor )
   struct UCB *fdf_ucb;
   uint_32 pdrv_no, pattern;
   uint_16 cylinder, head, fmt_type, interleave_factor;
```

```
uint_32 _Make_logical_drive( fdf_ucb, pdrv_no, ldrv_no, num_blks, blk_size,
                             cyl_offset, label )
   struct UCB *fdf_ucb;
   uint_32 pdrv_no, ldrv_no, num_blks, blk_size, cyl_offset;
   char *label;
```

**Datagram interface to TCP/IP network server**

```
uint_32 _TCP_abort( ucb )
   struct UCB *ucb;

uint_16 _TCP_awaitclose( ucb )
   struct UCB *ucb;

uint_32 _TCP_close( ucb )
   struct UCB *ucb;

struct TCPMSG *_TCP_msgrecv( ucb, msg, size )
   struct UCB *ucb;
   struct TCPMSG *msg;
   uint_16 size;

uint_16 _TCP_receive( ucb, msg, size )
   struct UCB *ucb;
   struct TCPMSG *msg;
   uint_16 size;

uint_32 _TCP_send( ucb, msg, size, controls, timeout )
   struct UCB *ucb;
   struct TCPMSG *msg;
   uint_16 size;
   uint_16 controls;
   uint_16 timeout;

struct TCPMSG *_TCP_waitmsg( ucb, msg, event )
   struct UCB *ucb;
   struct TCPMSG *msg;
   uint_16 event;
```

# APPENDIX D. SUPPLIED TASKS

**Kernel**

| | |
|---|---|
| _Directory() | root function for the directory task used to establish connections between clients and servers |
| _Gossip() | root function for the general logging and reporting task, used in particular to report processor faults |
| _Explicit_Scheduler() | root function for a scheduling task that uses a recurring interrupt to define epochs, and to facilitate other tasks being run after n epochs, as well as defining the order in which tasks waiting for an epoch will run |

**Debugger**

| | |
|---|---|
| _Dbg_control() | root function for the debug control server that manages planted breakpoints |
| _Dbg_agent() | root function for the debug agent task which is instantiated on each target processor to perform operations on behalf of the debug control task |
| _Dbg_shadow() | root function for the debug shadow task which provides timeouts for the debug control server for operations being performed by the debug agent tasks |

**Clock**

| | |
|---|---|
| _Clock_server() | root function for an alarm clock task that replies after a specified interval or at a specified time, and allows a waiting task to be explicitly awakened, as well as facilitating reading and setting of time |
| _Clock_notifier() | root function for a task receiving interrupts from a timer |

**Terminal**

| | |
|---|---|
| _Ehvt_server() | root function for an echo handshake virtual terminal server |
| _Ehvtic() | root function for the EHVT input consumer task that empties the input buffer and sends the characters to the server |
| _Ehvtip() | root function for the EHVT input producer task that fills the input buffer with bytes received from an ACIA |
| _Tty_server() | root function for a terminal server |
| _Tti() | root function for a task receiving bytes from an ACIA |
| _Tto() | root function for a task transmitting bytes to an ACIA |

**File system**

| | |
|---|---|
| _File_system_server() | root function for a *smart peripheral* task that uses the multiple disjoint linear address spaces on logical disks implemented by a _File_device_server to provide a tree structured file system with access controls |
| _File_device_server() | root function for a *smart peripheral* task that uses the raw sectors, tracks, cylinders and spindles of a typical disk controller to provide multiple disjoint linear address spaces on logical disks |
| _FD_Format_server() | root function for a *smart peripheral* task identical to the _File_device_server except that it also allows formatting of the underlying device. This task is neither necessary nor safe to use with an ordinary application program, but is needed when new media need to be initialized, or damaged media need to be repaired. |
| _FD_Seek_notifier() | root function for a task to notify the _File_device_server that a seek has completed |
| _FD_Cmd_notifier() | root function for a task to notify the _File_device_server that a disk command has completed |

**TCP/IP network**

| | |
|---|---|
| _TCP_server() | root function for the transmission control protocol server |
| _TCP_open_server() | root function for the transmission control protocol open server |
| _TCPIP_timer() | task to interface between the clock server and network tasks |
| _IP_server() | root function for the internet protocol server |
| _IP_courier() | task to relay messages for the server |
| _IP_reassembly() | root function for the internet datagram fragment reassembly task |
| _Tx_NI3010() | root function of a task that issues commands to the NI3010A ethernet communications controller |
| _Rx_NI3010() | root function of a task that receives a buffer from the NI3010A ethernet communications controller |

# APPENDIX E. SOFTWARE ENGINEERING CONSIDERATIONS: SOURCE MANAGEMENT AND THE ORGANIZATION OF THE HARMONY SOURCE TREE

The problem of managing Harmony source is different from that addressed by most software development tools, such as Make, or SCCS, or RCS on Unix. Those tools are intended to manage a program as it evolves. The concern is that as the program changes with time two things may happen:

(1) parts of the program on which other parts depend may change, necessitating recompilation, redoing linkage editing, or other processing, in order to rebuild a correct and consistent version of the program. This is what Make addresses.

(2) conflicting requirements of production use and further development dictate a regime of a sequence of frozen versions of the program, termed releases, which production users can count on as being stable while developers continue to evolve the subsequent release. The frozen versions may be expected to have minor changes as errors are repaired, leading to subreleases. If the program is big, and worked on by several developers, they may need to be protected from tripping over each other, for instance by preventing accidental and incompatible changes to the same module. These problems are what SCCS (Source Code Control System) and RCS (Revision Control System) address.

The software management problem facing Harmony is completely different, and although the two other problems exist they are minor by comparison. The problem of Harmony is that it is not one program, but a family of programs. The problem arises because Harmony is portable, configurable, and open, and because development is supported in many different computing environments. This has the following implications for source code:

(1) It is not known what computer system will be used for development of Harmony itself or application programs to be used in Harmony. The first consequence is that though the source for Harmony is organized in a tree, the form of pathnames identifying a particular unit cannot be stated in general! Another consequence is that the tools available are in general unknown, much less how the tools are invoked - unless they are tools provided with Harmony itself. Finally, the mechanism for executing command sequences on a given development system is unknown.

(2) It is not known what target microprocessor the Harmony system will be for. Care in using portable techniques means that this is surprisingly insignificant. Of course the functions written in assembler need redoing for a new microprocessor, but apart from that, the main things dependent on instruction set are the data structures describing the registers and other information saved on the stack when a task is interrupted, and for the same reason the information hand-crafted onto the stack when a task is created.

(3) It is not known what target microcomputer the target microprocessor is used in. This can affect what resources are available, what interrupt priority levels have been preallocated, how interprocessor interrupts are generated, what addresses are preallocated and what are available (what the memory map must be), what powerup sequence and other initialization is required, etc. Both assembly language and C code can be affected.

(4) It is not known what compiler and assembler will be used on the source. The compiler matters because there are minor differences in the languages accepted. With care, this can generally be avoided, but a major difference is that Whitesmiths', or any other compiler

based on C as in Unix version 6 or earlier, does not permit the casting of external initializers to match the type of the variable being initialized — whereas compilers based on C as in Unix version 7 or later require the casting for type conformance. The assembler matters rather more, as the syntax of different assemblers may be quite different. Register and subroutine linkage assumptions made by the compiler may require different assembly language code. Field offsets in structs must be calculated by hand for the assembly code, as assemblers rarely interpret C header files. Most annoying, the external symbols generated by C compilers rarely are exactly the identifiers used in C programs, so the assembly code must be changed to generate the appropriate symbols itself - the Unisoft compiler not even being consistent across different installations!

(5) The pathetic linkers and library editors used in the industry often are incapable of resolving backward references to modules occurring earlier in a library but not already forced to be loaded. This means that a unit of compilation bigger than a function must be used, so all such references can occur within a single linkage module.

(6) Limits on table sizes in various compilers mean that the unit of compilation must be limited to be smaller than the total source. In particular, many compilers will not accept all the header files as input in a single compilation, because the number of structs and unions, the number of members of structs and unions, or the number of defines, exceeds the maximum for which table space exists.

(7) The intended market of board level computers means that not only is the specific configuration unknown, e.g., device addresses, but the actual devices can differ in functionality and interface. A Motorola 6850 ACIA is vaguely equivalent to a Signetics 2681 DUART or an Intel 8251A USART, but the detail differences do affect the code for a terminal server. Similarly, the differences among timer chips or alarm clock chips do affect the code for the clock server. Peripherals implemented as separate boards, such as disk controllers, also have differences that are not transparent to the code for servers, such as the file device server, and might be used with any microcomputer implemented for the same bus. However, for a complex server, most of the code of the server is independent of these differences, and should be common across versions.

(8) Since many realtime systems are dedicated or highly specialized, Harmony is both configurable and open, in that any part of the supplied software not needed can be omitted, and customized servers required for the application can readily be added, whether they are modifications of supplied servers or something wholly new. Different modes of system use must be supported also, from programs downloaded into target systems for execution, to programs linked with Harmony then burned in ROM, to programs linking to a silicon operating system version of Harmony itself supplied in ROM.

(9) Many application programs must have access to header files from Harmony itself, i.e., data structure definitions and manifest constant values. This is true both of programs intended to run with Harmony on the target environment, and tools intended to run on whatever development environment is used to produce Harmony programs. For the latter, problems such as byte-swapping must be kept in mind.

These sorts of issues call for a new kind of source management. The solution used with Harmony evolved from the solution used with Thoth to a similar (but simpler) problem. It is to regard the Harmony source tree as a database of source code, from which appropriate items are extracted to make the version of interest at the moment. The granularity of the database is that one function is stored in each file, although there are also files containing logically grouped external declarations, and logically grouped sets of #define, struct and union definitions.

The extraction is done by files containing only #include statements, and since the extraction is for a specific version, file pathnames can be expressed in the appropriate format. An *inclusion file*, as these are known, defines a compilation unit which should be compiled together so the resulting object module can be stored in a library without the ordering concerns mentioned above. The source explicitly makes no use of the #ifdef conditional compilation mechanism, first because experience shows that if many alternate versions of code are combined this way the code becomes unreadable, and second because finding code with version differences then requires searching the complete text of the code, rather than just identifying files. The tree structure of the file system is heavily used in organizing the database.

The first branching of the Harmony source directory identifies major categories of items. These include the system itself (sys), the supplied tools (tools), example programs (example), documentation (doc), and command files (com). The tools and example subdirectories are divided at the next level into specific tools and specific examples. The command files subdirectory is subdivided at the next level by specific development environments, such as VMS or MCS.

For a specific program or library, such as /Harmony/example/srtest/ or /Harmony/sys/, the next directory level distinguishes the source code database (src) from the directories of inclusion files for compiling various versions of this entity. Any version has a corresponding directory (with possible subdirectories) of inclusion files, the structure in general paralleling the structure (described below) of the source directory. At the moment the following versions are supplied in the tree: ch00wvms which is for compiling under VMS on the DEC VAX using Whitesmiths' compiler with the target being an Omnibyte OB68K1A Chorus machine, ch10wvms which is the same except that a 68010 processor replaces the 68000 processor on the OB68K1A, ch20wvms which is the same except that a 68020 replaces the 68000 processor on the OB68K1A, and ch00umcs which is for compiling under MCS on the Wicat using the Unisoft compiler with the target also being an Omnibyte OB68K1A Chorus machine, ch10umcs being the same except for the 68010, and ch20umcs being the same except for the 68020. There are also versions present for the Ikonas (Adage 3000) workstation compiling under VMS on the DEC VAX using Whitesmiths' compiler for each of the 68000, 68010, and 68020. Documentation about the specific program or library, if provided, is under a directory (doc) also at this level.

The source code directory (src) may contain files of source code directly. On the other hand, if the program or library is made up of major configurable abstractions, it will be divided into subdirectories corresponding to these abstractions, and only things like header files needed for all abstractions will appear at this level. Abstraction subdirectories themselves may have subdirectories if the abstraction subdivides naturally into subabstractions, as sys/src/servers/ does. The source code directly under src/ or directly under an abstraction such as sys/src/connect/ is only that code in C which is identical for all versions. If some attribute dictates a special variant, another level of subdirectory named for that attribute is interposed above the source code. Clearly the contents below different variant-discriminating subdirectories are parallel to each other. Similarly, when the code is written in assembler instead of C, a subdirectory is interposed naming the specific assembler. This is not completely satisfactory, as it implies an assembler is used with only one compiler. Every variant subdirectory contains a file variant.doc describing what characterizes the need for that variant.

An inclusion directory should contain a file version.doc describing exactly what version this directory corresponds to, i.e., what variants are included. Under an inclusion directory, there will be subdirectories corresponding to the abstraction subdirectories (if any) under the

corresponding source directory. There are no variant subdirectories, as these would correspond to different inclusion directories. As yet there are no language subdirectories, in part because these too would correspond to different inclusion directories, and in part because as yet none of the assembiers we have used can cope with #include files - in fact, most of them insist on producing the relocatable object module in the same subdirectory where the assembler source was found. Primarily, the files under the inclusion directory are inclusion files, to be compiled. Some inclusion files are provided merely to generate listings, as the Listings tool works on inclusion files. Our normal working procedure is to set the current directory to somewhere in the inclusion directory, and compile the inclusion files there, with the result that the corresponding object module appears at that point in the inclusion directory. The object modules are then combined into either a library, or an executable program image, that is placed immediately under the top of the inclusion directory. Related files, such as load maps, are also placed where the library or executable image is.